**Báo Cáo Tổng Hợp: Triển Khai Nền Tảng Phân Quyền (Authorization)**

Mục tiêu chính của các bước này là thiết lập một hệ thống nơi người dùng có thể được gán các vai trò (như "Admin" hoặc "User"), và sau đó các vai trò này có thể được sử dụng để kiểm soát quyền truy cập vào các tài nguyên hoặc chức năng cụ thể trong API .

**1. Xây Dựng Cấu Trúc Dữ Liệu cho Phân Quyền (Models & DbContext)**

Để hỗ trợ hệ thống vai trò, cần thêm các thực thể mới vào cơ sở dữ liệu và định nghĩa mối quan hệ giữa chúng.

* **Model Role.cs**:
  + Đã tạo một bảng riêng biệt để lưu trữ định nghĩa các vai trò (ví dụ: "Admin", "User"). Mỗi vai trò có một Id và Name.

public class Role

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

[Required]

[MaxLength(50)]

public string Name { get; set; } = string.Empty;

// qh vơis UserRole (n-n)

public ICollection<UserRole>? UserRoles { get; set; }

}

}

* **Model UserRole.cs (Bảng trung gian)**:
  + Đã tạo một bảng nối (join table) UserRole để quản lý mối quan hệ nhiều-nhiều (many-to-many) giữa User và Role. Một người dùng có thể có nhiều vai trò, và một vai trò có thể được gán cho nhiều người dùng.
  + Mỗi bản ghi trong UserRole chứa UserId và RoleId.

public class UserRole

{

// Khóa ngoại to tb User

[ForeignKey("User")]

public int UserId { get; set; }

public User User { get; set; } = null!;

// Khóa ngoại to tb Role

[ForeignKey("Role")]

public int RoleId { get; set; }

public Role Role { get; set; } = null!;

}

* **Cập nhật ApplicationDbContext.cs**:
  + Đã thêm DbSet cho Role và UserRole để Entity Framework Core có thể tương tác với các bảng này.
  + Trong phương thức OnModelCreating, đã cấu hình mối quan hệ nhiều-nhiều giữa User và Role thông qua UserRole. Điều này bao gồm định nghĩa khóa chính tổ hợp (HasKey) và mối quan hệ khóa ngoại (HasOne, WithMany, HasForeignKey, OnDelete).
  + **Seed Data (Dữ liệu ban đầu)**: đã thêm dữ liệu mặc định cho bảng Roles với các vai trò "Admin" và "User" ngay khi database được tạo thông qua migration.

modelBuilder.Entity<Role>().HasData(

new Role { Id = 1, Name = "Admin" },

new Role { Id = 2, Name = "User" }

);

* **Migration và Database Update**: Đã chạy các lệnh Add-Migration và Update-Database để áp dụng những thay đổi này vào cơ sở dữ liệu, tạo ra các bảng Roles và UserRoles mới.

**Phân tích:** Việc thiết lập ba bảng (Users, Roles, UserRoles) với mối quan hệ nhiều-nhiều là cách chuẩn và mạnh mẽ để triển khai hệ thống phân quyền dựa trên vai trò. Nó cung cấp sự linh hoạt cao, cho phép bạn dễ dàng thêm các vai trò mới và gán chúng cho người dùng mà không cần thay đổi cấu trúc bảng User.

**2. Tích Hợp Vai Trò vào Quy Trình Xác Thực (Authentication)**

Để sử dụng các vai trò đã gán, cần đưa thông tin vai trò vào token xác thực của người dùng.

* **Cập nhật phương thức Login trong AuthController.cs**:
  + Khi người dùng đăng nhập thành công, ta đã thêm logic để **lấy tất cả các vai trò** mà người dùng đó được gán từ bảng UserRoles.
  + Các vai trò này được thêm vào danh sách Claims của người dùng (claims.Add(new Claim(ClaimTypes.Role, userRole.Role.Name))).
  + Sau đó, các Claims này được sử dụng để tạo **JWT (JSON Web Token)**. Khi JWT được phát hành, nó sẽ chứa thông tin về vai trò của người dùng.
* **Cấu hình JWT Authentication trong Program.cs**:
  + đã cấu hình AddJwtBearer để xác thực JWT. Khi một yêu cầu có kèm JWT được gửi đến API, middleware xác thực sẽ đọc token và trích xuất các Claim (bao gồm các vai trò) từ đó.

**Phân tích:** Việc nhúng vai trò vào JWT là một phương pháp phổ biến và hiệu quả. Khi client gửi một JWT hợp lệ, API có thể dễ dàng trích xuất thông tin vai trò mà không cần phải truy vấn lại database, giúp giảm tải cho server và cải thiện hiệu suất.

**3. Áp Dụng Phân Quyền (Authorization) cho API Endpoints**

Đây là bước cuối cùng để thực sự sử dụng các vai trò để kiểm soát quyền truy cập.

* **Sử dụng thuộc tính [Authorize(Roles = "RoleName")]**:
  + Trong AdminController.cs, chúng ta đã thêm thuộc tính [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme, Roles = "Admin")] ở cấp Controller.
  + Điều này có nghĩa là **tất cả các endpoint** trong AdminController (ví dụ: GetUsersWithRoles, AssignRole, RemoveRole, DisableUser, EnableUser, AdminUpdateUser) **chỉ có thể được truy cập bởi người dùng đã đăng nhập và có vai trò "Admin"**.
  + Nếu một người dùng không phải Admin cố gắng truy cập các endpoint này, họ sẽ nhận được phản hồi HTTP 403 Forbidden.

**Phân tích:** Thuộc tính [Authorize] là cách đơn giản và hiệu quả nhất trong ASP.NET Core để thực hiện phân quyền. Bằng cách chỉ định các vai trò cần thiết, bạn có thể kiểm soát chi tiết ai có thể truy cập từng phần của API. Việc đặt nó ở cấp Controller là cách nhanh chóng để bảo vệ toàn bộ nhóm các chức năng quản trị.

**Triển Khai Các API Quản lý Người dùng trong AdminController.cs**:

* **GET /api/Admin/users-with-roles**: Cho phép Admin lấy danh sách tất cả người dùng cùng với các vai trò mà họ đang có.
* **POST /api/Admin/assign-role**: Admin có thể gán một vai trò hiện có cho một người dùng cụ thể.
* **POST /api/Admin/remove-role**: Admin có thể hủy gán một vai trò hiện có khỏi một người dùng.
* **GET /api/Admin/roles**: Admin có thể lấy danh sách tất cả các loại vai trò hiện đang có trong hệ thống (ví dụ: "Admin", "User").

Báo Cáo Cập Nhật và Phân Tích Tính Năng Mới

**1. Cập Nhật Bảng User với Trạng Thái IsActive**

**Mục đích:** Để quản lý trạng thái hoạt động của tài khoản người dùng, cho phép vô hiệu hóa mà không cần xóa vĩnh viễn dữ liệu.

**Thay đổi chính:**

* **Model User.cs**: Đã thêm thuộc tính public bool IsActive { get; set; } = true;. Giá trị mặc định là true để các tài khoản mới được tạo ra sẽ ở trạng thái hoạt động.
* **Migration và Database Update**: Lệnh Add-Migration AddIsActiveToUser và Update-Database đã được sử dụng để thêm cột IsActive vào bảng Users trong cơ sở dữ liệu. Điều này đảm bảo cơ sở dữ liệu vật lý khớp với định nghĩa model của bạn.
* **Logic trong AuthController.cs**:
  + **Phương thức Login**: Đã thêm điều kiện kiểm tra if (!user.IsActive) ngay sau khi xác thực mật khẩu. Nếu người dùng không hoạt động, API sẽ trả về Unauthorized với thông báo "Tài khoản của bạn đã bị vô hiệu hóa." Điều này ngăn chặn người dùng bị vô hiệu hóa đăng nhập vào hệ thống.

**Phân tích:**

Việc thêm IsActive là một phương pháp quản lý tài khoản linh hoạt hơn so với việc xóa. Nó giúp **bảo toàn dữ liệu lịch sử** liên quan đến người dùng (như các giao dịch, bài viết, v.v.) ngay cả khi tài khoản của họ không còn được sử dụng. Admin có thể dễ dàng kiểm soát quyền truy cập và kích hoạt lại tài khoản khi cần, điều này hữu ích trong nhiều tình huống nghiệp vụ (ví dụ: đình chỉ tài khoản tạm thời, tài khoản bị khóa do vi phạm, hoặc khôi phục tài khoản đã lâu không dùng).

**2. Triển Khai Tính Năng Quản Lý Người Dùng bởi Admin (AdminController.cs)**

**Mục đích:** Cung cấp các công cụ cho người dùng có vai trò "Admin" để quản lý các tài khoản người dùng khác.

**Thay đổi chính:**

* **DTO AdminUserUpdateDto**: Đã định nghĩa DTO này để chứa thông tin người dùng có thể được Admin cập nhật (UserId, Username, Email, và IsActive). Việc sử dụng DTO giúp kiểm soát dữ liệu đầu vào và validation.
* **Các Endpoint mới trong AdminController.cs**:
  + **[HttpPost("disable-user/{userId}")] DisableUser(int userId)**: Nhận userId và cập nhật IsActive của người dùng đó thành false. Có kiểm tra để đảm bảo người dùng tồn tại và chưa bị vô hiệu hóa.
  + **[HttpPost("enable-user/{userId}")] EnableUser(int userId)**: Nhận userId và cập nhật IsActive của người dùng đó thành true. Có kiểm tra để đảm bảo người dùng tồn tại và chưa được kích hoạt.
  + **[HttpPut("update-user")] AdminUpdateUser([FromBody] AdminUserUpdateDto dto)**: Cho phép Admin cập nhật Username, Email và **IsActive** của người dùng. Đặc biệt, nó kiểm tra tính duy nhất của Username mới và chỉ cập nhật các trường có giá trị được cung cấp. Phương thức này **không** cho phép Admin thay đổi mật khẩu trực tiếp, duy trì tính bảo mật.

**Phân tích:**

Các API này được bảo vệ bởi thuộc tính [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme, Roles = "Admin")], đảm bảo rằng **chỉ người dùng có vai trò "Admin" và đã xác thực bằng JWT** mới có thể thực hiện các thao tác này. Điều này rất quan trọng để duy trì quyền riêng tư và an toàn dữ liệu. Việc phân tách các chức năng quản trị vào AdminController giúp cấu trúc code rõ ràng và dễ quản lý quyền truy cập.

**3. Triển Khai Tính Năng Reset Mật khẩu (Forgot Password)**

**Mục đích:** Cho phép người dùng đặt lại mật khẩu của họ khi quên, sử dụng một quy trình an toàn dựa trên email và token.

**Thay đổi chính:**

* **Model PasswordResetToken.cs**: Một bảng mới đã được tạo để lưu trữ các token reset mật khẩu.
  + Các thuộc tính quan trọng: UserId, Token (chuỗi duy nhất), CreatedAt, ExpiresAt (thời gian hết hạn), và IsUsed (đánh dấu token đã được sử dụng để tránh tấn công replay).
* **Cập nhật ApplicationDbContext.cs**:
  + Thêm DbSet<PasswordResetToken> để tương tác với bảng mới.
  + Cấu hình mối quan hệ khóa ngoại từ PasswordResetToken đến User trong OnModelCreating, với OnDelete(DeleteBehavior.Cascade) để tự động xóa token khi người dùng bị xóa. Đã thêm Unique Index cho cột Token để đảm bảo tính duy nhất.
* **DTOs mới trong AuthDTOs.cs**:
  + **ForgotPasswordRequestDto**: Chứa Email của người dùng yêu cầu reset.
  + **ResetPasswordConfirmDto**: Chứa Email, Token và NewPassword (cùng với ConfirmNewPassword để xác nhận) cho quá trình xác nhận.
* **Dịch vụ Email (IEmailService và EmailService.cs)**:
  + **Thiết kế Interface (IEmailService)**: Giúp tách biệt logic gửi email khỏi các controller, làm cho code dễ kiểm thử và thay đổi (ví dụ: chuyển đổi giữa các nhà cung cấp dịch vụ email).
  + **Triển khai (EmailService.cs)**: Sử dụng thư viện MailKit và MimeKit để tương tác với SMTP server. Đọc thông tin cấu hình từ appsettings.json.
  + **Cấu hình appsettings.json**: Đã thiết lập các thông số SMTP (SmtpServer, SmtpPort, SenderEmail, SenderName, SenderPassword, MailtrapPassword) để kết nối với Mailtrap, một công cụ tuyệt vời để kiểm thử email trong môi trường phát triển mà không gửi email thật.
  + **Đăng ký EmailService trong Program.cs**: Sử dụng builder.Services.AddTransient<IEmailService, EmailService>() để .NET Core có thể tự động inject IEmailService vào các lớp cần nó.
* **Các Endpoint mới trong AuthController.cs**:
  + **[HttpPost("forgot-password")]**:
    - Tìm người dùng theo email.
    - Tạo Guid làm token và thiết lập thời gian hết hạn.
    - Lưu token vào database và xóa các token cũ chưa sử dụng của người dùng đó.
    - Xây dựng resetLink (quan trọng: trong production, đây sẽ là URL của frontend của bạn).
    - Gọi \_emailService.SendEmailAsync() để gửi email.
    - **Điểm bảo mật**: Luôn trả về phản hồi chung chung ("Nếu email tồn tại...") để tránh tiết lộ thông tin về việc email có tồn tại hay không.
  + **[HttpPost("reset-password-confirm")]**:
    - Tìm người dùng và token reset mật khẩu.
    - **Xác minh chặt chẽ token**: Kiểm tra tính hợp lệ của token, xem nó đã được sử dụng hay chưa và còn hạn sử dụng không.
    - Nếu token hợp lệ, băm mật khẩu mới (BCrypt.Net.BCrypt.HashPassword) và cập nhật PasswordHash của người dùng.
    - Đánh dấu token là IsUsed = true để ngăn chặn việc sử dụng lại.
* **Phân tích:**Tính năng reset mật khẩu là một thành phần **bảo mật quan trọng**. Việc sử dụng token một lần, có thời hạn và được đánh dấu là đã sử dụng giúp ngăn chặn các cuộc tấn công replay hoặc sử dụng token hết hạn. Việc tách rời logic gửi email vào một dịch vụ riêng biệt cũng là một thực hành tốt, giúp code module hơn. Cấu hình Mailtrap cho môi trường phát triển là một quyết định thông minh, giúp quá trình phát triển nhanh chóng và dễ dàng kiểm thử hơn.

**Dự tính triển khai tiếp theo:**  
**Logging và Xử lý lỗi nâng cao**:

* + Tích hợp một framework logging mạnh mẽ (ví dụ: Serilog, NLog) để ghi lại các sự kiện và lỗi của ứng dụng.
  + Triển khai Middleware xử lý lỗi toàn cục (Exception Handling Middleware) để bắt và xử lý các ngoại lệ một cách tập trung, cung cấp phản hồi lỗi nhất quán cho client.

1. **Triển khai vào Container (Docker)**:
   * Đóng gói ứng dụng API vào Docker container để dễ dàng triển khai, quản lý và mở rộng trên các môi trường khác nhau (ví dụ: Kubernetes).
2. **Sử dụng Response Caching/Redis**:
   * Đối với các endpoint đọc dữ liệu thường xuyên được truy cập, có thể áp dụng caching để cải thiện hiệu suất và giảm tải cho cơ sở dữ liệu.